第33卷 第22期 计 算 机 工 程 2007年11月
Vol.33 No.22 Computer Engineering November 2007
·软件技术与数据库· 文章编号：1000—3428(2007)22—0076—04 文献标识码：A 中图分类号：TP391

基于AOP的面向对象程序的单元测试

张 艳，赵建军，冯 斌
(上海交通大学软件学院，上海 200240)

摘 要：单元测试被视为横切关注点，在软件测试过程中很重要。而面向方面编程(AOP)的主要目标就是分离横切关注点，因此，单元测试非常适合用AOP来解决。该文提出了在对面向对象程序进行单元测试时遇到的问题，并用AOP方法加以解决，同时比较了传统方法和AOP方法进行单元测试的优缺点。

关键词：面向方面编程；单元测试；AspectJ

Object-oriented Unit Test Based on Aspect-oriented Programming

ZHANG Yan, ZHAO Jian-jun, FENG Bin
(School of Software, Shanghai Jiaotong University, Shanghai 200240)

【Abstract】 Unit test is regarded as crosscutting concern, which is a very important testing behavior in the process of software test. However, the main objective of aspect-oriented programming(AOP) is to separate crosscutting concerns, so AOP is fit for unit test. This paper brings forward several problems existing in object-oriented unit testing and shows how the AOP can be used to solve them. Virtues and shortcomings of using traditional method and AOP methods to support object-oriented unit test are compared.

【Key words】 aspect-oriented programming(AOP); unit test; AspectJ

1 概述

单元测试是在软件测试过程中进行的最早期的、且非常重要的测试活动，它测试的对象是软件设计的最小单位——模块[1]。单元测试分为两类：(1)基于规范的单元测试(黑盒测试)，主要是根据需求文档验证软件组件的功能和行为；(2)基于程序的单元测试(白盒测试)，主要是检查程序的内部逻辑是否正确。软件测试的目的之一是尽早发现软件中存在的错误，降低软件质量成本。由于单元测试是早期发现错误的最好时机，因此它在软件开发过程中受到了极大重视。

面向方面编程(aspect-oriented programming, AOP)技术通过引入分离关注点形成模块化的机制来解决横切关注点(crosscutting concerns)问题。AspectJ[2]是Xerox PARC开发的基于Java语言的AOP扩展，AspectJ通过Java语言实现主要关注点，并通过对Java进行扩展的Aspect机制实现横切关注点。AspectJ向Java语言增加了几种特定的语言结构，见表1。

表1 增加的特定语言结构

术语	作用
join point(连接点)	程序执行中的特定的点，用以表明可以插入的横切行为的位置
pointcut(切入点)	用以捕捉程序执行中的特定连接点，并搜集该连接点上下文的程序结构
advice(通知)	预先定义好的在特定的连接点出现时要执行的代码
inter-type declaration (类型间声明)	一种向以前建立的类中添加属性及方法的强大机制
aspect(方面)	一种类似于Java类的结构，对连接点、pointcut，advice及类型间声明进行封装

单元测试代码有时需要在被测实现(implementation under test, IUT)中嵌入代码；有时为了测试更加完整，测试代码需要对私有成员进行特权访问。另外，单元测试代码通常在测试阶段结束之后被移除。因此，横切于组件中的单元测试可被视为相对于被测实现的横切关注点，而这十分适合用AOP技术来解决。

2 示例程序

示例程序是一个简化的银行系统，包含3类：BankAccount，AccountService及AccountManager。

代码1 BankAccount.java

```
package bank;
public class BankAccount {
  private double balance;
  private String owner;
  private double integral;
  public BankAccount(String owner,double initialBalance){
        this.setOwner(owner);
        this.deposit(initialBalance);}
  public double deposit(double amount){//存钱
        this.balance +=amount;
        return this.balance; }
  public double withdraw(double amount){//取钱
        this.balance -=amount;
        return this.balance; }
  //消费积分反馈
  public double integral(double rate, double amount){
        if(rate>0 && amount>100){//condition 1
              integral=amount*1.2;}
        if(rate>3 ||amount>3000){//condition 2
              integral=amount*1.5;}
        return integral;}
  public double getBalance() {
        return balance;}
```

作者简介：张 艳(1980－)，女，硕士研究生：主研方向：软件工程，面向方面编程；赵建军，教授、博士生导师；冯 斌，硕士研究生

收稿日期：2006-11-29 **E-mail：**emily_zhang@sjtu.edu.cn

```
    public void setBalance(double balance) {
        this.balance = balance;}
    public void setOwner(String owner) {
        this.owner = owner;}}
```

代码 1 表示一个有 3 个属性的 BankAccount 对象，该类实现 3 个功能：存钱，取钱，记录消费反馈积分。

代码 2 AccountManager.java

```
package bank;
import java.sql.*;
import java.util.*;
public class AccountManager {
    public BankAccount findAccountForUser(String userid){
        // some code logic to load a user account using JDBC
        double balance=0;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
          Connection connection =
          DriverManager.getConnection("jdbc:odbc:temp");
           String sql="select userid form Account";
           Statement statement=connection.createStatement();
           ResultSet result=statement.executeQuery(sql);
           while(result.next()){
                balance=result.getDouble("balance"); }
           result.close();
           statement.close();
           connection.close(); }
       catch(Exception e){
            e.printStackTrace(); }
       BankAccount ba=new BankAccount(userid,balance);
       return ba; }
    public void updateAccount(BankAccount account){
       //some code logic to update certain account }}
```

代码 2 管理 BankAccount 对象的持久性和生命周期，可以通过账户 ID 查找账户及更新账户。

代码 3 AccountService.java

```
package bank;
import java.util.*;
public class AccountService {
    private AccountManager accountManager;
    public void setAccountManager(AccountManager manager){
        this.accountManager=manager; }
    public void transfer(String senderId,String receiverId, double
amount){
        BankAccount sender
        =this.accountManager.findAccountForUser(senderId);
        BankAccount receiver
        =this.accountManager.findAccountForUser(receiverId);
        sender.withdraw(amount);
        receiver.deposit(amount);
        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(receiver); }}
```

代码 3 提供有关账户的服务，显示用于两个账户之间转账的 transfer 方法。

3 契约检查

由第 2 节中的银行系统代码可知，BankAccount类虽然简洁，仍存在许多问题，如deposit或withdraw方法的amount参数能否为负数或 0；账户余额能否为负数。对此，有 2 种解决方法：(1)采用在Java的 1.4 版本中引入的断言机制；(2)契约式设计。该设计是面向对象软件设计中的技术，目的是保证软件的质量、可靠性和可重用性[3]。它使用 3 类测试来指定和确保组件的行为：1)前置条件测试：测试组件的输入，指定组件执行请求的操作之前必须满足的条件。2)后置条件测试：在前置条件已满足的情况下，确保组件完成操作后的结果符合要求。3)不变式测试：测试永远不变的条件。如字段在对象的整个生命周期中保持不变。

可以使用 AspectJ 来直接测试组件的前置/后置条件及不变式。因为 advice(通知)可以在方法被调用之前和之后插入，这样就可以测试传入被调用方法的参数是否有效；测试方法被调用后是否返回了有效结果。

示例程序中的 BankAccount 接口规范包含的前置条件测试是：(1)要求客户传递给 withdraw 和 deposit 方法的参数大于或等于 0；(2)withdraw 方法还要求取款的数额不能超过现有余额；(3)withdraw 和 deposit 方法都含有相似的后置条件测试，即返回的值必须等于新余额，新余额必须等于旧余额减去或加上输入的钱的数额；(4)包含一个类范围的不变条件测试，即余额总要大于或等于 0。

前置条件测试由 before advice 处理，它在对应的方法执行连接点之前执行。如果测试失败，就抛出异常，报告错误消息，同时程序执行中断。前置条件的测试如代码 4 所示，在这个 aspect 中，定义了 2 个 pointcut，并分别定义了 before advice。第 1 个 advice 会检查存取钱的额度，并在这个数额小于 0 时抛出异常。第 2 个 advice 会检查取钱的额度，如果这个数额超过账户中的余额，同样会抛出异常。

代码 4 前置条件测试

```
package bank;
public aspect AccountPreconditionAspect {
    //存、取的钱数不能为负数
    pointcut preventNegativeAmounts(BankAccount account,
double amount)
      : (execution(* BankAccount.deposit(double))
        || execution(* BankAccount.withdraw(double))) &&
this(account) && args(amount);
    //取的钱数不能超过账户的余额(不能透支)
    pointcut preventOverdraft(BankAccount account, double
amount)
    :   execution(* BankAccount.withdraw(double)) && this(account)
&& args(amount);
    before(BankAccount account, double amount)
      :preventNegativeAmounts(account, amount) {
        if (amount < 0)
            throw new RuntimeException("Negative amounts
not permitted"); }
    before(BankAccount account, double amount)
      :preventOverdraft(account, amount) {
        if (account.getBalance() < amount)
          throw new RuntimeException("Insufficient funds"); }}
```

后置条件测试由 after advice 来处理，该建议在其作用的连接点之后执行。代码 5 对 BankAccount 接口规范中的后置条件进行测试，也定义了两个 pointcut，并分别定义了 after advice，以确保执行方法 deposit 和 withdraw 后，得到正确的返回值。

代码 5 后置条件测试

```
package bank;
public aspect AccountPostConditionAspect {
    pointcut assureDepositeBalance(BankAccount account, double
```

```
amount)
        :call (* BankAccount.deposit(double)) && this(account) &&
args(amount);
        pointcut assureWithdrawBalance(BankAccount account,
double amount)
        :call(* BankAccount.withdraw(double)) && this(account)&&
args(amount);
            after(BankAccount account, double amount):
            assureWithdrawBalance(account,amount){
            assertBalance(account.getBalance()-amount ==
account.withdraw(amount)); }
        after(BankAccount account, double amount):
            assureDepositeBalance(account, amount){
            assertBalance(account.getBalance()+amount ==
account.deposit(amount)); }
        private void assertBalance(boolean v) {
            if( !v )
            throw new RuntimeException("check balance"); }}
```

不变条件测试使用 around advice，该建议在连接点执行之前和之后都可以运行。代码 6 对 BankAccount 接口规范中的不变条件进行了测试，如果账户中的余额大于或等于 0，就返回该值，否则就返回-0.0，用来表示错误的余额。

代码 6 不变式测试

```
package bank;
public aspect AccountInvariantAspect {
  double around(BankAccount account): target(account) &&
call(* BankAccount.getBalance()) {
        double balance=proceed(account);
        System.out.println("balance="+balance);
        if(balance>=0.0){
            return balance;}
        else{
            return -0.0;} }}
```

采用 AspectJ 中的 advice 机制进行契约检查优于 Java1.4 版本中加入的断言机制，因为：(1)测试完成之后，可以移走这些测试 aspect，关闭契约式设计测试，消除它们带来的开销；(2)前置/后置条件、不变条件测试的方面代码与应用源代码的实现是分开的。这就使得相同的方面测试代码可以用于多个地方(如果在其他地方有意义的话)；(3)如果软件说明书写得非常充分，契约检查可作为黑盒测试机制，给测试人员带来便利。如果产品在投入使用后出现问题，这些方面代码又可以很方便地加入到应用软件中，给产品支持人员提供附加信息。

因此，AspectJ 可以用于为系统测试和单元测试实现前置/后置条件测试和不变式测试。

4 私有成员测试

在面向对象系统中，属性和方法都封装在类中，只能通过预留的接口进行访问。但在对程序进行单元测试时，不仅需要对模块的接口进行测试，还可能需要测试模块局部数据结构。这样，封装就会引起对被测试类的不充足访问。

若不使用外部工具，用传统的方法解决这个问题需要为测试而改变代码。测试人员可以用 public 关键字代替所有的 private 关键字，然后进行测试。但这样做的缺点是必须维护实现相同功能的两个版本的代码，增加了维护的难度和成本。

当然，也可以用 AOP 方法来解决这个问题。AspectJ 提供了关键字 privileged，将 aspect 定义为特权的，特权 aspect 可以访问类的私有属性。在示例程序 BankAccount 类中有 3 个私有属性：balance，owner 和 integral。由于 balance 和 integral 属性都提供了一个简单的 get 访问方法，因此属性值很容易获得。而 owner 属性没有定义 get 方法，所以，外部对象就不能访问 owner 属性(在实际编码时，该方式不会被采用，这里为了更好地说明问题)。为了在测试时能够访问 owner 属性，可以定义一个特权 aspect，如代码 7 所示。这种方法的优点是不需要对源代码做任何修改，不会额外增加代码维护的开销。

代码 7 PrivateAttributeAspect.aj

```
package bank;
public privileged aspect PrivateAttributeAspect {
    pointcut accessPrivateAttribute(BankAccount account):
    initialization(BankAccount.new(..))&& target(account);
    after(BankAccount account)
      :accessPrivateAttribute(account){
      System.out.println("account.owner="+account.owner);}}
```

添加在 aspect 上的 privileged 关键字允许 aspect 对私有属性拥有内部访问权限，因此，可以利用 AspectJ 这一特点，解决在对类进行测试时由于封装所引起的不完全访问问题，从而方便地测试模块局部数据结构，并且不需要对源代码做任何修改，便于维护。

5 孤立单元测试

孤立单元测试很好，如：可以测试还未写完的代码，替代难以实现的环境等。为了进行隔离测试，Tim Mackinnon 等首先提出了模仿对象(mock objects，简称mocks)的概念[4]。Mocks替换测试中与所用的方法协作的对象，从而提供隔离层，但它不实现任何逻辑。

传统的单元测试中用模仿来进行隔离测试，但在模仿对象策略时，将真正的类换成 mock 有时很困难。因此，代码经常要修改，以提供暗门。由于 AOP 的概念之一就是拦截特殊的方法调用，并取代其内容，这与模仿对象的功能非常类似，因此本文采用 AOP 方法来隔离单元测试。在 AspectJ 中，可以利用 around advice 拦截方法调用。

如对示例程序中的 AccountService.transfer 方法执行单元测试，但在该方法的内部调用了 AccountManger.findAccount ForUser 方法，而这个方法需要使用 JDBC 与后台数据库建立连接，这时就需要隔离测试 transfer 方法。如代码 8 所示，为根据账户 ID 查找账户的方法返回一个模仿的 BankAccount 对象，替代真实的方法调用后返回的结果。

代码 8 MockObjectAspect.aj

```
package bank;
public aspect MockObjectAspect {
  pointcut inTest():execution(public void *.*());
  pointcut mockObject(String userid)
 :cflow(inTest())&&call(public BankAccount
*.findAccountForUser(String))&& args(userid);
   Object around(String userid):mockObject(userid){
        BankAccount ba = new BankAccount(userid, 500);
        return ba;}
```

这个例子虽然很简单，但它的确可以替代模仿对象来实现隔离单元测试，并且在有些案例中采用 mock 遇到困难时，这种方法仍然适用。另外，可以将每个模仿对象都用一个 aspect 来实现，并用 around 通知来替代连接点的执行，不会影响要测试的类。所以，AOP 方法不仅可以解决模仿对象遇

到的问题(如需要修改代码提供暗门)，且便于维护。仅仅简单地采用 around 来拦截方法调用并不能体现出这种方法的优势所在。文献[5]提出了利用 AOP 技术进行隔离单元测试的框架，并已经被证明在多层系统中开发单元测试非常有效。

6 探测覆盖率

单元测试中的白盒测试针对程序内部的逻辑结构来设计，用逻辑覆盖率衡量测试的完整性。逻辑覆盖有语句覆盖、判定覆盖、条件覆盖与路径覆盖等。

在设计白盒测试用例时，本文先完成黑盒测试，然后统计白盒覆盖率，针对未覆盖的逻辑单位设计测试用例覆盖它，这样既检验了黑盒测试的完整性，又避免了重复工作；传统的方法是利用额外的测试覆盖率工具来统计覆盖率，这就需要额外花费时间和精力来学习如何使用这些工具，导致开发时间的延长。

在采用 AOP 方法探测覆盖率时，由于 AspectJ 提供了非常细致的跟踪功能和日志功能，因此，可以通过跟踪方法的调用和记录日志来确定调用了哪个方法，执行了哪些语句，从而获得测试覆盖率。如在示例代码 BankAccount 类中的 integral 方法，包含 2 个条件语句，分别命名为 Condition1 和 Condition2，每个条件又都有 2 个分支，分别命名为 branchA、branchB、branchC 及 branchD。探测覆盖率的方面测试代码如代码 9 所示，在此，本文定义了 3 个通知：第 1 个 before 通知在 integral 方法执行之前记录日志，显示进入该方法的执行和执行的分支；第 2 个 after 通知在 integral 方法执行之后记录日志，显示退出该方法的执行；第 3 个 before 通知在为变量赋值时记录日志，显示变量 integral 的新值。

代码 9 TraceAspect.aj

```
package bank;
import java.util.logging.*;
import org.aspectj.lang.*;
public aspect TraceAspect {
  private static final int branchA = 0,branchB = 100,branchC =
3,branchD = 3000;
   private Logger _logger = Logger.getLogger("trace");
   pointcut traceMethods(double rate, double amount)
  :execution (public double BankAccount.integral(..)) &&!within
(TraceAspect)&&args(rate,amount);
   before(double rate, double amount)
  : traceMethods(rate,amount) {
  Signature sig = thisJoinPointStaticPart.getSignature();
  _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
  sig.getName(), "-->Entering");
        if (rate > branchA)
            _logger.logp(Level.INFO, "" +
thisJoinPoint.getSignature(),"Location:" +
thisJoinPoint.getSourceLocation(),"Branch A is True!");
        if (amount > branchB)
            _logger.logp(Level.INFO, "" +
thisJoinPoint.getSignature(),"Location:" +
thisJoinPoint.getSourceLocation(),"Branch B is True!");
        if (rate > branchC)
            _logger.logp(Level.INFO, "" +
thisJoinPoint.getSignature(),"Location:" +
thisJoinPoint.getSourceLocation(),"Branch C is True!");
        if (amount > branchD)
            _logger.logp(Level.INFO, "" +
thisJoinPoint.getSignature(),"Location:" +
thisJoinPoint.getSourceLocation(),"Branch D is True!");}
   after(double rate, double amount)
  :traceMethods(rate,amount){
        Signature sig = thisJoinPointStaticPart.getSignature();
        _logger.logp(Level.INFO,
sig.getDeclaringType().getName(), sig.getName(), "<--Exit");}
   before(double n)
  :set(double BankAccount.integral)&&args(n){
        _logger.logp(Level.INFO,"==Setting >:" +
thisJoinPoint.getSignature(), "Location:"+
thisJoinPoint.getSourceLocation(),"Integral New Value:" + n);}}
```

运行程序后，控制台显示的信息如下所示：

```
2006-10-8 21:16:37 bank.BankAccount integral
信息: -->Entering
2006-10-8 21:16:37 double bank.BankAccount.integral(double,
double) Location:BankAccount.java:18
信息: Branch A is True!
2006-10-8 21:16:37 double bank.BankAccount.integral(double,
double) Location:BankAccount.java:18
信息: Branch C is True!
2006-10-8 21:16:37 ==Setting >:double
bank.BankAccount.integral Location:BankAccount.java:23
信息: Integral New Value:15.0
2006-10-8 21:16:37 bank.BankAccount integral
信息: <--Exit
```

由控制台信息中可知，哪些语句和哪些路径已经执行，取到了分支判断和条件分支的哪些值等(如这里分支判断 Condition1 取到了 False，Condition2 取到了 True；条件分支 branchA 取到了 True，branchB 取到了 False，branchC 取到了 True，branchD 取到了 False)。从而，通过分析所记录的日志，就能探测到测试用例的覆盖率。

因此，AspectJ 中的日志和跟踪功能可以用于探测单元测试的覆盖率，其优点是不需要额外学习新工具的使用，简单方便。

7 结束语

AspectJ 是 Java 语言的一种面向 AOP 的扩展，在传统的面向对象方法失败的地方或不是特别理想的地方代之以 test-only 行为，从而用其他方法进行单元测试。本文提出了几个在对面向对象程序进行单元测试时可能会遇到的问题，并用 AOP 方法进行解决，且分析了其优点。对 AOP 所拥有的功能来说，单元测试是非常小的一部分应用，在将来还应进一步挖掘 AOP 丰富而强大的功能。

参考文献

1 朱少民. 软件测试方法和技术[M]. 北京: 清华大学出版社, 2005.

2 AspectJ. AspectJ-Homepage[Z]. (2006-05). http://www.aspectj.org. 2006.

3 Diotalevi F. 用 AOP 增强契约[Z]. (2004-11). http://www-128.ibm.com/developerworks/cn/java/j-ceaop/.

4 Mackinnon T, Freeman S, Craig P. Endo-Testing: Unit Testing with Mock Objects[C]//Proc. of eXtreme Programming and Flexible Processes in Software Engineering Conference, Sardinia, Italy. 2000: 21-23.

5 Monk S, Virtual H S. Mock Objects Using AspectJ with JUNIT[Z]. (2002-04). http://www.xprogramming.com/xpmag/virtualMockObjects.htm.